JAVA 9
Java Platform Module System
(JPMS)

Java Platform Module System (JPMS)

Introduction:

Modularity concept introduced in Java 9 as the part of Jigsaw project. It is the main important concept in java 9.

The development of modularity concept started in 2005.
The First JEP(JDK Enhancement Proposal) for Modularity released in 2005. Java people tried to release Modularity concept in Java 7(2011) & Java 8(2014). But they failed. Finally after several postponements this concept introduced in Java9.

Until Java 1.8 version we can develop applications by writing several classes, interfaces and enums. We can places these components inside packages and we can convert these packages into jar files. By placing these jar files in the classpath, we can run our applications. An enterprise application can contain 1000s of jar files also.

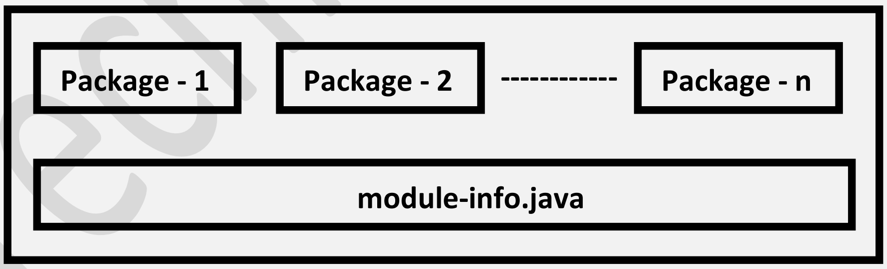
Hence jar file is nothing but a group of packages and each package contains several .class files.

But in Java 9, a new construct got introduced which is nothing but 'Module'. From java 9 version onwards we can develop applications by using module concept.

Module is nothing but a group of packages similar to jar file. But the specialty of module when compared with jar file is, module can contain configuration information also.

Hence module is more powerful than jar file. The configuration information of module should be specified in a special file named with module-info.java

Every module should compulsory contains module-info.java, otherwise JVM won't consider that as a module of Java 9 platform.



In Java 9, JDK itself modularized. All classes of Java 9 are grouped into several modules (around 98) like

- java.base
- java.logging
- java.sql
- java.desktop(AWT/Swing)
- java.rmi etc

java.base module acts as base for all java 9 modules.

We can find module of a class by using getModule() method.

Eg:

```
System.out.println(String.class.getModule());//module java.base
```

What is the need of JPMS?

Application development by using jar file concept has several serious problems.

Problem-1: Unexpected *NoClassDefFoundError* in middle of program execution

There is no way to specify jar file dependencies until java 1.8V.At runtime,if any dependent jar file is missing then in the middle of execution of our program, we will get NoClassDefFoundError, which is not at all recommended.

Demo Program to demonistrate NoClassDefFoundError:

```
durgajava9
  |-A1.java
  |-A2.java
  |-A3.java
  |-Test.java
```

A1.java:

```java
package pack1;
public class A1
{
    public void m1()
    {
        System.out.println( "pack1.A" );
    }
}
```

A2.java

```java
package pack2;
import pack1.A1;
public class A2
{
    public void m2()
    {
        System.out.println( "pack2.A2  method" );
        A1 a = new A1();
        a.m1();
    }
}
```

A3.java:

```
import  pack2.A2;
class Test
{
    public  static  void  main(String[]  args)
    {
        System.out.println(   "Test   class   main" );
        A2  a= new  A2();
        a.m2();
    }
}
```

```
D:\durgajava9>javac -d . A1.java
D:\durgajava9>javac -d . A2.java
D:\durgajava9>javac -d . A3.java
D:\durgajava9>javac Test.java
```

```
durgajava9
  |-Test.class
  |-pack1
     |-A1.class
  |-pack2
     |-A2.class
```

At runtime, by mistake if pack1 is not available then after executing some part of the code in the middle, we will get NoClassDefFoundError.

```
D:\durgajava9>java Test
Test class main
pack2.A2 method
Exception in thread "main" java.lang.NoClassDefFoundError: pack1/A1
```

But in Java9, there is a way to specify all dependent modules information in module-info.java. If any module is missing then at the beginning only, JVM will identify and won't start its execution. Hence there is no chance of raising *NoClassDefFoundError* in the middle of execution.

Problem 2: Version Conflicts or Shadowing Problems

If JVM required any .class file, then it always searches in the classpath from left to right until required match found.

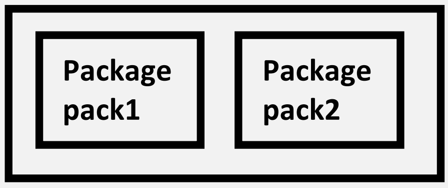
classpath = jar1;jar2;jar3;jar4

If jar4 requires Test.class file of jar3.But Different versions of Test.class is available in jar1, jar2 and jar3. In this case jar1 Test.class file will be considered, because JVM will always search from Left to Right in the classpath. It will create version conflicts and causes abnormal behavior of program.

But in java9 module system, there is a way to specify dependent modules information for every module seperately.JVM will always consider only required module and there is no order importance. Hence version conflicts won't be raised in Java 9.

<u>Problem 3:</u> Security problem

There is no mechanism to hide packages of jar file.



Jar File

Assume pack1 can be used by other jar files, but pack2 is just for internal purpose only.
Until Java 8 there is no way to specify this information. Everything in jar file is public and available to everyone. Hence there may be a chance of Security problems.

public is too much public in jar files.

But in Java 9 Module system, we can export particular package of a module. Only this exported package can be used by other modules. The remaining packages of that module are not visible to outside. Hence Strong encapsulation is available in Java 9 and there is no chance of security problems.

Even though class is public, if module won't export the corresponding package, then it cannot be accessed by other modules. Hence public is not really that much public in Java 9 Module System.

Module can offer Strong Encapsulation than Jar File.

<u>Problem 4:</u> JDK/JRE having Monolithic Structure and Very Large Size

The number of classes in Java is increasing very rapidly from version to version.

JDK 1.0V having 250+ classes
JDK 1.1V having 500+ classes
...
JDK 1.8V having 4000+ classes

And all these classes are available in rt.jar.
Hence the size of rt.jar is increasing from version to version.
The size of rt.jar in Java 1.8Version is around 60 MB.

To run small program also, total rt.jar should be loaded, which makes our application heavy weight and not suitable for IOT applications and micro services which are targeted for portable devices.

It will create memory and performance problems also.
(This is something like inviting a Big Elephant in our Small House: Installing a Heavy Weight Java application in a small portable device).

But in java 9, rt.jar removed. Instead of rt.jar all classes are maintained in the form of modules. Hence from Java 9 onwards JDK itself modularized. Whenever we are executing a program only required modules will be loaded instead of loading all modules, which makes our application light weighted.

Now we can use java applications for small devices also. From Java 9 version onwards, by using JLINK , we can create our own very small custom JREs with only required modules.

Q. Explain differences between jar file and java 9 module

Jar	Module
1) Jar is a group of packages and each package contains several classes	1) Module is also a group of packages and each package contains several classes. Module can also contain one special file module-info.java to hold module specific dependencies and configuration information.
2) In jar file, there is no way to specify dependent jar files information	2) For every module we have to maintain a special file module-info.java to specify module dependencies
3) There is no way to check all jar file dependencies at the beginning only. Hence in the middle of the program execution there may be a chance of NoClassDefFoundError.	3) JVM will check all module dependencies at the beginning only with the help of module-info.java. If any dependent module is missing then JVM won't start its execution. Hence there is no chance of NoClassDefFoundError in the middle of execution.
4) In the classpath the order of jar files important and JVM will always considers from left to right for the required .class files. If multiple jars contain the same .class file then there may be a chance of Version conflicts and results abnormal behavior of our application	4) In the module-path order is not important. JVM will always check from the dependent module only for the required .class files. Hence there is no chance of version conflicts and abnormal behavior of the application.
5) In jar file there is no mechanism to control access to the packages. Everything present in the jar file is public to everyone. Any person is allowed to access any component from the jar file. Hence there may be a chance of security problems	5) In module there is a mechanism to control access to the packages. Only exported packages are visible to other modules. Hence there is no chance of security problems
6) Jars follows monolithic structure and applications will become heavy weight and not suitable for small devices.	6) Modules follow distributed structure and applications will become light weighted and suitable for small devices.
7) Jar files approach cannot be used for IOT devices and micro services.	7) Modules based approach can be used for IOT devices and micro services

Q. What is Jar Hell or Classpath Hell?

The problems with use of jar files are:

1. *NoClassDefFoundError* in the middle of program execution
2. Version Conflicts and Abnormal behavior of program
3. Lack of Security
4. Bigger Size

This set of Problems is called Jar Hell OR Classpath Hell. To overcome this, we should go for JPMS.

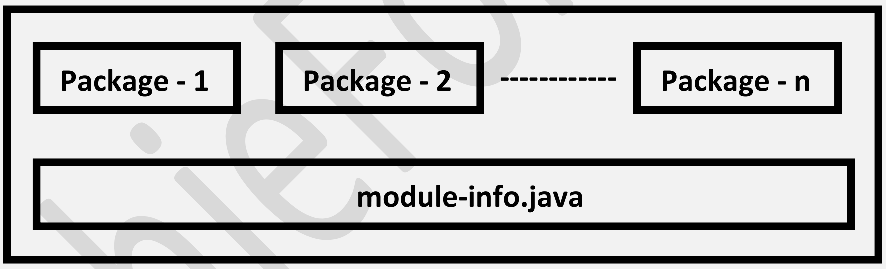
Q. What are various Goals/Benefits of JPMS?

1. Reliable Configuration
2. Strong Encapsulation & Security
3. Scalable Java Platform
4. Performance and Memory Improvements

etc

What is a Module:

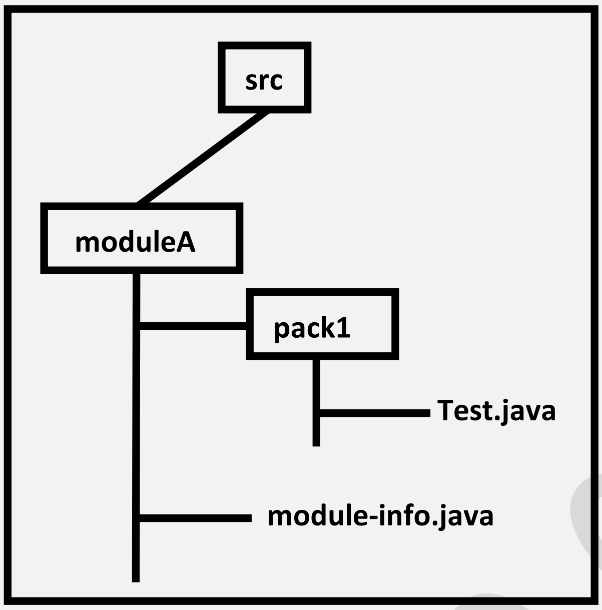
Module is nothing but collection of packages. Each module should compulsory contains a special configuration file: module-info.java.



We can define module dependencies inside module-info.java file.

```
module moduleName
{
  Here we have to define module dependencies which represents
  1. What other modules required by this module?
  2. What packages exported by this module for other modules?
  etc
}
```

Steps to Develop First Module Based Application:



Step-1: Create a package with our required classes

```
package pack1;
public class Test
{
    public static void main(String[] args)
    {
        System.out.println( "First Module in JPMS");
    }
}
```

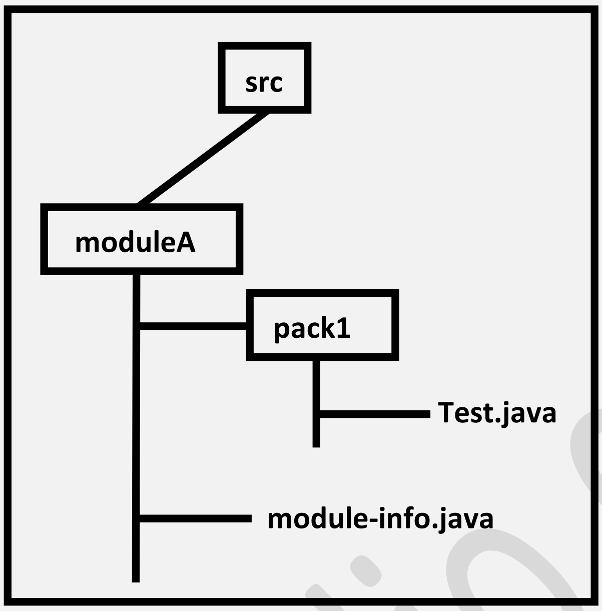
Step-2: Writing module-info.java

For every module we should to write a special file named with module-info.java.
In this file we have to define dependencies of module.

```
module moduleA
{
}
```

Step-3: Arrange all files in the required package structure

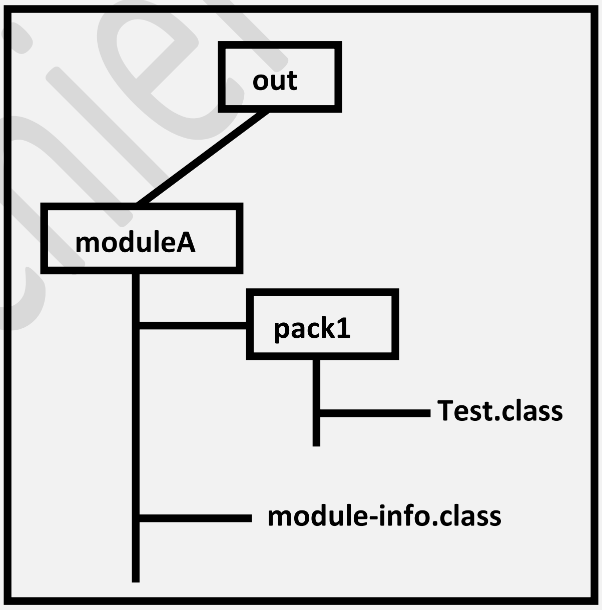
Arrange all the files according to required folder structure



Step-4: Compile module with --module-source-path option

```
javac --module-source-path src -d out -m moduleA
```

The generated class file structure is:



Step-5: Run the class with --module-path option

```
java --module-path out -m moduleA/pack1.Test
```

Output: First Module in JPMS

Case-1:

If module-info.java is not available then the code won't compile and we will get error. Hence module-info.java is mandatory for every module.

```
javac --module-source-path src -d out -m moduleA
error: module moduleA not found in module source path
```

Case-2:

Every class inside module should be part of some package, otherwise we will get compile time error saying : unnamed package is not allowed in named modules
In the above application inside Test.java if we comment package statement

```java
//package pack1;
```

```java
public class Test
{
    public static void main(String[] args)
    {
        System.out.println("First Module in JPMS");
    }
}
```

error: unnamed package is not allowed in named modules

Case-3:

The module name should not ends with digit(like module1,module2 etc),otherwise we will get warning at compile time.

```
javac --module-source-path src -d out -m module1
warning: [module] module name component module1 should avoid terminal digits
```

Various Possible Ways to Compile a Module:

```
javac --module-source-path src -d out -m moduleA
javac --module-source-path src -d out --module moduleA
javac --module-source-path src -d out
          src/moduleA/module-info.java src/moduleA/pack1/Test.java
javac --module-source-path src -d out
       C:/Users/Durga/Desktop/src/moduleA/module-info.java
C:/Users/Durga/Desktop/src/moduleA/pack1/Test.java
```

Various Possible Ways to Run a Module:

```
java --module-path out --add-modules moduleA pack1.Test
java --module-path out -m moduleA/pack1.Test
java --module-path out --module moduleA/pack1.Test
```

Inter Module Dependencies:

Within the application we can create any number of modules and one module can use other modules.

We can define module dependencies inside module-info.java file.

```
module moduleName
{
  Here we have to define module dependencies which represents
  1. What other modules required by this module?
  2. What packages exported by this module for other modules?
  etc
}
```

Mainly we can use the following 2 types of directives

1. requires directive:

It can be used to specify the modules which are required by current module.

Eg:

```
module  moduleA
{
    requires   moduleB;
}
```

It indicates that moduleA requires members of moduleB.

Note:

1. We cannot use same requires directive for multiple modules. For every module we have to use separate requires directive.

requires moduleA,moduleB; ➔ invalid

2. We can use requires directive only for modules but not for packages and classes.

2. exports directive:

It can be used to specify what packages exported by current module to the other modules.

Eg:

```
module  moduleA
{
    exports  pack1;
}
```

It indicates that moduleA exporting pack1 package so that this package can be used by other modules.

Note: We cannot use same exports directive for exporting multiple packages. For every package a separate exports directive must be required.

exports pack1,pack2; ➔ invalid

Note: Be careful about syntax requires directive always expecting module name where as exports directive expecting package name.

```
module  modulename
{
    requires  modulename;
    exports  packagename;
}
```

Note:

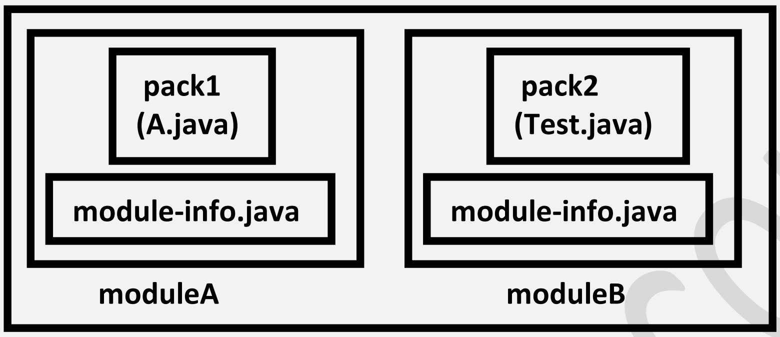
By default all packages present in a module are private to that module. If module exports any package only that particular package is accessible by other modules. Non exporting packages cannot be accessed by other modules.

Eg: Assume moduleA contains 2 packages pack1 and pack2. If moduleA exports only pack1 then other modules can use only pack1. pack2 is just for its internal purpose and cannot be accessed by other modules.

```
module  moduleA
{
    exports  pack1;
}
```

Demo program for inter module dependencies:

Rectangle diagram which represents total application



moduleA components:

A.java:

```
package  pack1;
public  class  A
{
    public  void  m1()
    {
        System.out.println(  "Method  of  moduleA" );
    }
}
```

module-info.java:

```
module  moduleA
{
    exports  pack1;
}
```

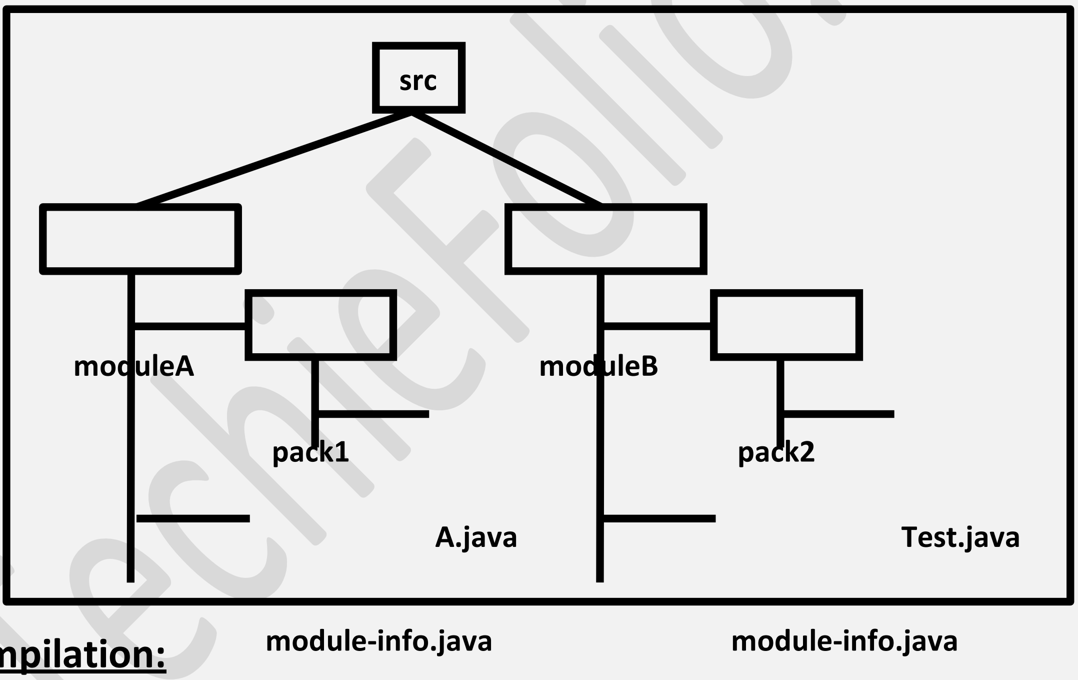
moduleB components:

Test.java:

```
package pack2;
 import pack1.A;
 public class Test
{
    public static void main(String[] args)
    {
        System.out.println( "moduleB accessing members of moduleA" );
        A a = new A();
        a.m1();
    }
 }
```

module-info.java:

```
module moduleB
{
    requires moduleA;
}
```

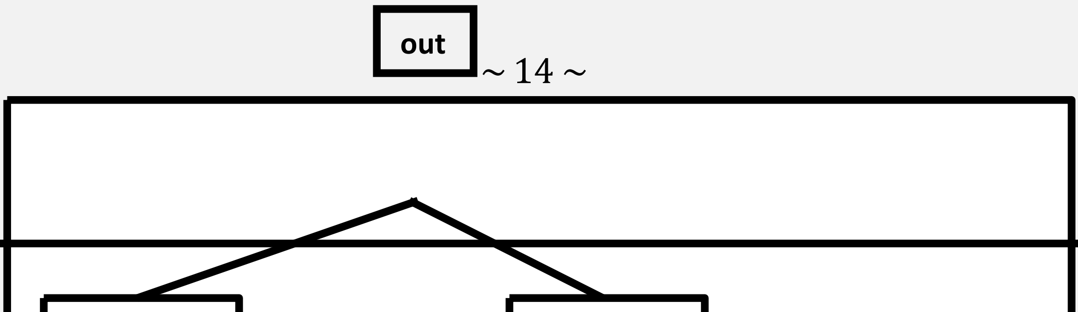


Compilation:

C:\Users\Durga\Desktop>javac --module-source-path src -d out -m moduleA,moduleB

<u>Note:</u> space is not allowed between the module names otherwise we will get error.

```
C:\Users\Durga\Desktop>javac --module-source-path src -d out -m moduleA, moduleB
error: Class names, 'moduleB', are only accepted if annotation processing is explicitly requested
```



Execution:

```
C:\Users\Durga\Desktop>java --module-path out -m moduleB/pack2.Test
```

Output:

```
moduleB accessing members of moduleA
Method of moduleA
```

Case-1:

Even though class A is public, if moduleA won't export pack1, then moduleB cannot access A class.

Eg:

```
module  moduleA
{
    //exports   pack1;
}
```

```
C:\Users\Durga\Desktop>javac --module-source-path src -d out -m moduleA,moduleB
src\moduleB\pack2\Test.java:2: error: package pack1 is not visible
import pack1.A;
     ^
  (package pack1 is declared in module moduleA, which does not export it)
1 error
```

Case-2:

We have to export only packages. If we are trying to export modules or classes then we will get compile time error.

Eg-1: exporting module instead of package

```
module  moduleA
{
    exports  moduleA;
}
```

```
C:\Users\Durga\Desktop>javac --module-source-path src -d out -m moduleA, moduleB
src\moduleB\pack2\Test.java:2: error: package pack1 is not visible
import pack1.A;
     ^
  (package pack1 is declared in module moduleA, which does not export it)
src\moduleA\module-info.java:3: error: package is empty or does not exist: moduleA
      exports moduleA;
```

In this case compiler considers moduleA as package and it is trying to search for that package.

Eg-2: exporting class instead of package:

```
module  moduleA
{
    exports  pack1.A;
}
```

```
C:\Users\Durga\Desktop>javac --module-source-path src -d out -m moduleA, moduleB
src\moduleB\pack2\Test.java:2: error: package pack1 is not visible
import pack1.A;
     ^
  (package pack1 is declared in module moduleA, which does not export it)
src\moduleA\module-info.java:3: error: package is empty or does not exist: pack1.A
      exports pack1.A;
                   ^
2 errors
```

In this case compiler considers pack1.A as package and it is trying to search for that package.

Case-3:

If moduleB won't use "requires moduleA" directive then moduleB is not allowed to use members of moduleA, even though moduleA exports.

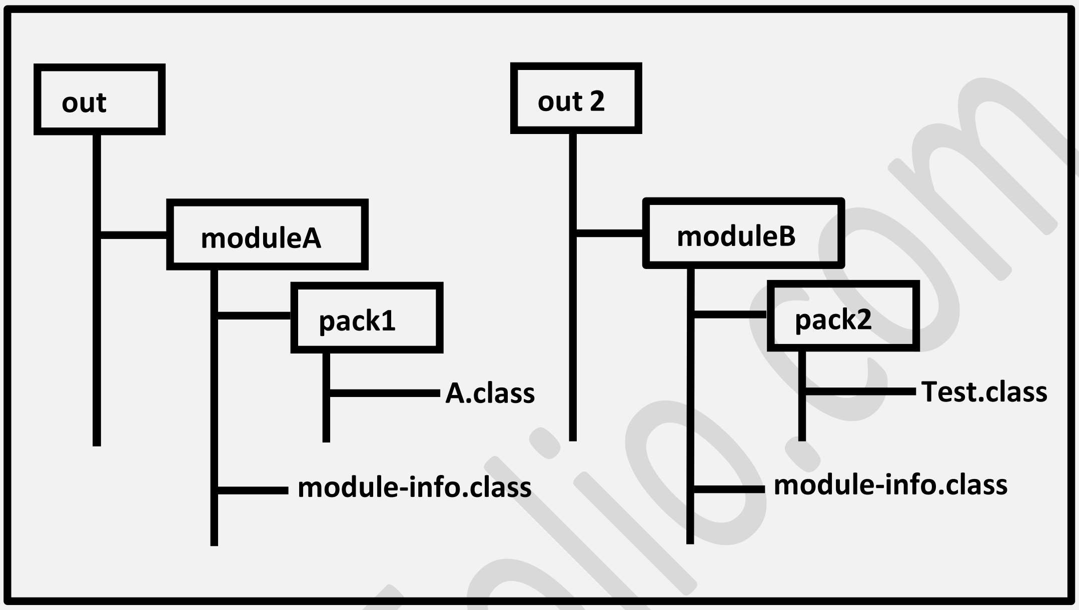
```
module  moduleB
{
    //requires   moduleA;
}
```

```
C:\Users\Durga\Desktop>javac --module-source-path src -d out -m moduleA, moduleB
src\moduleB\pack2\Test.java:2: error: package pack1 is not visible
import pack1.A;
     ^
  (package pack1 is declared in module moduleA, but module moduleB does not read it)
1 error
```

Case-4:

If compiled codes are available in different packages then how to run?
We have to use special option: --upgrade-module-path

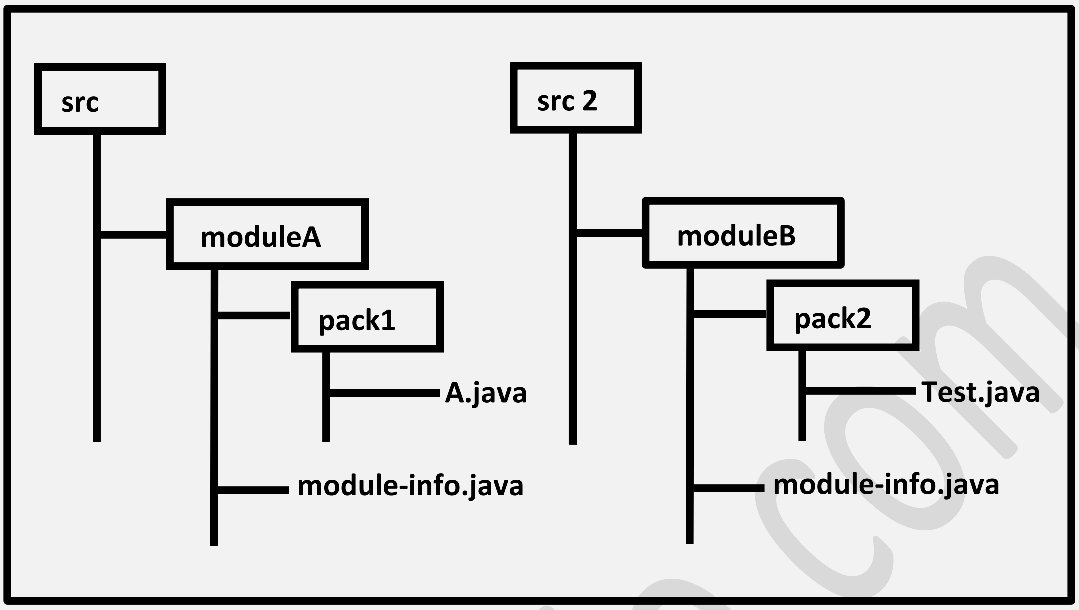
If compiled codes of moduleA is available in out and compiled codes of moduleB available in out2



```
C:\Users\Durga\Desktop>java --upgrade-module-path out;out1 -m moduleB/pack2.Test
moduleB accessing members of moduleA
Method of moduleA
```

Case-5:

If source codes of two modules are in different directories then how to compile?
Assume moduleA source code is available in src directory and moduleB source code is available in src2



```
C:\Users\Durga\Desktop>javac --module-source-path src;src2 -d out -m moduleA, moduleB
```

Q. Which of the following are meaningful?

```
module  moduleName
{
    1. requires  modulename;
    2. requires  modulename.packagename;
    3. requires  modulename.packagename.classname;
    4. exports   modulename;
    5. exports   packagename;
    6. exports   packagename.classname;
}
```

Answer: 1 & 5 are Valid

Note: We can use exports directive only for packages but not modules and classes, and we can use requires directive only for modules but not for packages and classes.

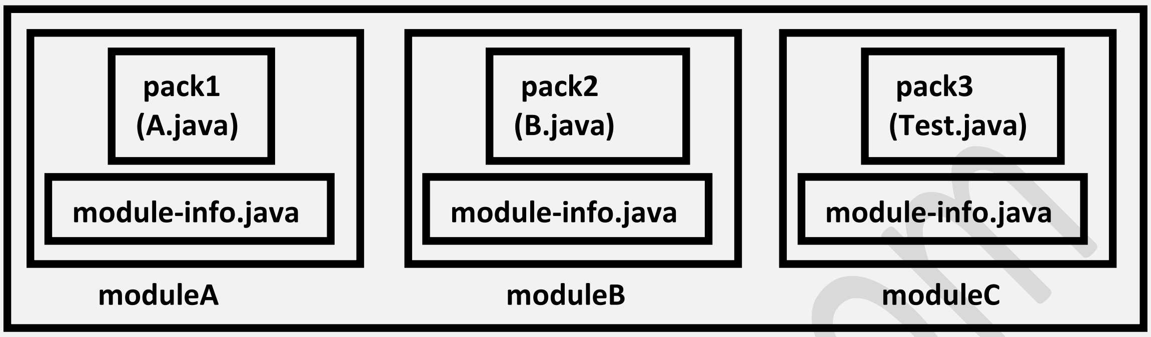
Note: To access members of one module in other module, compulsory we have to take care the following 3 things.

1. The module which is accessing must have requires dependency
2. The module which is providing functionality must have exports dependency
3. The member must be public.

JPMS vs NoClassDefFoundError:

In Java 9 Platform Modular System, JVM will check all dependencies at the beginning only. If any dependent module is missing then JVM won't start its execution. Hence there is no chance of NoClassDefFoundError in the middle of Program execution.

Demo Program:



Components of moduleA:

A.java:

```java
package pack1;
public class A
{
    public void m1()
    {
        System.out.println( "Method of moduleA" );
    }
}
```

module-info.java:

```java
module moduleA
{
    exports pack1;
}
```

Components of moduleB:

B.java:

```java
package pack2;
import pack1.A;
public class B
{
    public void m2()
    {
        System.out.println( "Method of moduleB" );
        A a = new A();
        a.m1();
    }
}
```

module-info.java:

```java
module moduleB
{
    requires moduleA;
    exports pack2;
}
```

Components of moduleC:

Test.java:

```java
package pack3;
import pack2.B;
public class Test
{
    public static void main(String[] args)
    {
        System.out.println( "Test class main method" );
        B b = new B();
        b.m2();
    }
}
```

module-info.java:

```java
module moduleC
{
    requires moduleB;
}
```

Compilation and Execution:

```
C:\Users\Durga\Desktop>javac --module-source-path src -d out -m moduleA, moduleB, moduleC
```

```
C:\Users\Durga\Desktop>java --module-path out -m moduleC/pack3.Test
Test class main method
Method of moduleB
Method of moduleA
```

If we delete compiled code of module (inside out folder), then JVM will raise error at the beginning only and JVM won't start program execution.

```
C:\Users\Durga\Desktop>java --module-path out -m moduleC/pack3.Test
Error occurred during initialization of boot layer
java.lang.module. FindException: Module moduleA not found, required by moduleB
```

But in Non Modular programming, JVM will start execution and in the middle, it will raise NoClassDefFoundError.

Hence in Java Platform Module System, there is no chance of getting NoClassDefFoundError in the middle of program execution.

Transitive Dependencies (requires with transitive Keyword):

A→B, B→C ==> A→C
This property in mathematics is called Transitive Property.

Student1 requires Material, only for himself, if any other person asking he won't share.

```
module  student1
{
    requires  material;
}
```

"Student1 requires material not only for himself, if any other person asking him, he will share it"

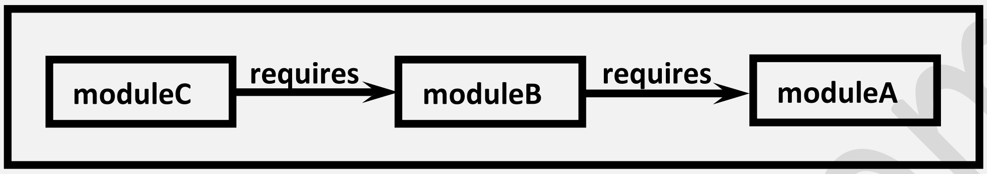
```
module  Student1
{
    requires  transitive   material;
}
```

Sometimes module requires the components of some other module not only for itself and for the modules that requires that module also. For this requirement we can use transitive keyword.

The transitive keyword says that "Whatever I have will be given to a module that asks me."

Case-1:

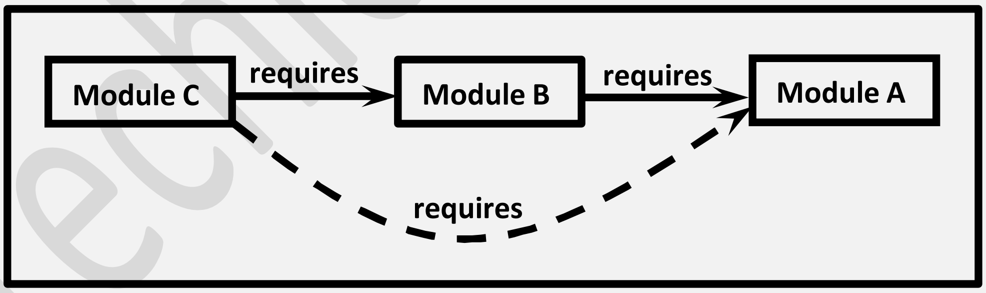
```
module  moduleA
{
    exports  pack1;
}
module  moduleB
{
    requires  moduleA;
}
module  moduleC
{
    requires  moduleB;
}
```



In this case only moduleB is available to moduleC and moduleA is not available. Hence moduleC cannot use the members of moduleA directly.

Case-2:

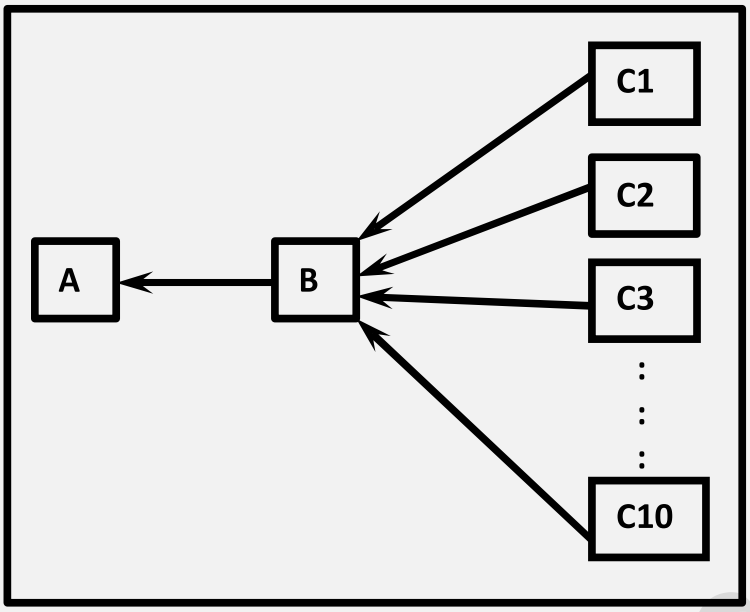
```
module  moduleA
{
    exports  pack1;
}
module  moduleB
{
    requires  transitive   moduleA;
}
module  moduleC
{
    requires  moduleB;
}
```



In this both moduleB and moduleA are available to moduleC. Now moduleC can use members of both modules directly.

Case Study:

Assume Modules C1, C2, ... C10 requires Module B and Module B requires A.



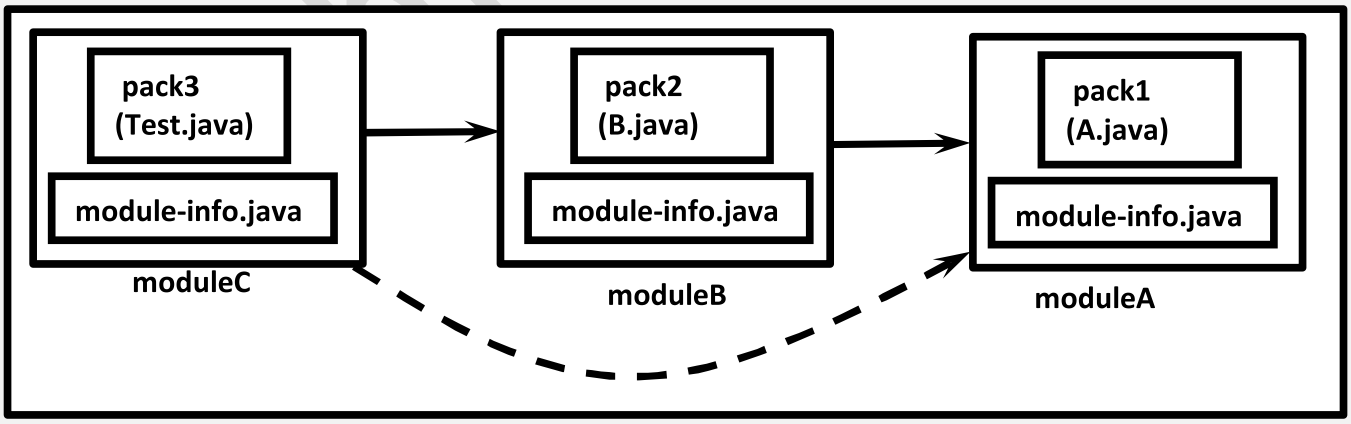
If we write "requires transitive A" inside module B

```
module  B
{
    requires  transitive   A;
}
```

Then module A is by default available to C1, C2,.., C10 automatically. Inside every module of C, we are not required to use "requires A" explicitly. Hence transitive keyword promotes code reusability.

Note: Transitive means implied readability i.e., Readability will be continues to the next level.

Demo Program for transitive keyword:



A.java:

```
package pack1;
public class A
{
    public  void  m1()
    {
        System.out.println(  "moduleA  method" );
    }
}
```

module-info.java:

```
module  moduleA
{
    exports  pack1;
 }
```

moduleB components:

B.java:

```
 package  pack2;
 import  pack1.A;
public  class   B
{
    public  A m2()
    {
        System.out.println(  "moduleB  method" );
        A a = new  A();
        return   a;
    }
}
```

module-info.java:

```
 module  moduleB
{
    requires  transitive   moduleA;
    exports  pack2;
 }
```

moduleC components:

Test.java:

```
package  pack3;
import  pack2.B;
public  class  Test
{
    public  static  void  main(String[]  args)
    {
        System.out.println( "Test  class  main  method" );
        B b = new  B();
        b.m2().m1();
    }
}
```

module-info.java:

```
module  moduleC
{
    requires  moduleB;
}
```

```
C:\Users\Durga\Desktop>javac --module-source-path src -d out -m moduleA, moduleB, moduleC

C:\Users\Durga\Desktop>java --module-path out -m moduleC/pack3.Test
Test class main method
moduleB method
moduleA method
```

In the above program if we are not using transitive keyword then we will get compile time error because moduleA is not available to moduleC.

```
javac --module-source-path src -d out -m moduleA,moduleB,moduleC
src\moduleC\pack3\Test.java:9: error: A.m1() in package pack1 is not accessible
            b.m2().m1();
                ^
 (package pack1 is declared in module moduleA, but module moduleC does not read it)
```

Optional Dependencies (Requires Directive with static keyword):

If Dependent Module should be available at compile time but optional at runtime, then such type of dependency is called Otional Dependency. We can specify optional dependency by using static keyword.

Syntax: requires static <modulename>

The static keyword is used to say that, "This dependency check is mandatory at compile time and optional at runtime."

Eg1:

```
module  moduleB
{
   requires  moduleA;
}
```

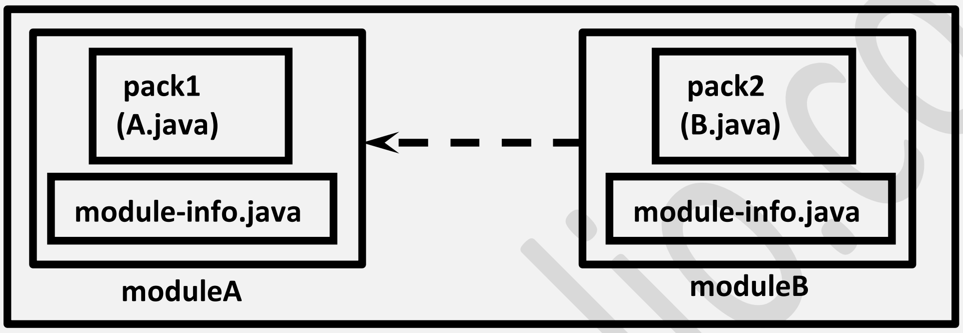
moduleA should be available at the time of compilation and runtime. It is not optional dependency.

Eg2:

```
module  moduleB
{
     requires  static  moduleA;
 }
```

At the time of compilation moduleA should be available, but at runtime it is optional. i.e., at runtime even moduleA is not available JVM will execute code.

Demo Program for Optional Dependency:



moduleA components:

A.java:

```
package  pack1;
public  class   A
{
     public  void  m1()
     {
         System.out.println(   "moduleA   method" );
     }
}
```

module-info.java:

```
module  moduleA
{
     exports  pack1;
 }
```

moduleB components:

Test.java:

```
package  pack2;
public  class   Test
{
    public  static  void  main(String[]   args)
    {
        System.out.println(  "Optional   Dependencies   Demo!!!" );

    }
}
```

module-info.java:

```
module  moduleB
{
    requires  static   moduleA;
}
```

At the time of compilation both modules should be available. But at runtime, we can run moduleB Test class, even moduleA compiled classes are not available i.e., moduleB having optional dependency with moduleA.

```
C:\Users\Durga\Desktop>javac --module-source-path src -d out -m moduleA,moduleB
```

```
C:\Users\Durga\Desktop>java --module-path out -m moduleB/pack2.Test
Optional Dependencies Demo!!!
```

If we remove static keyword and at runtime if we delete compiled classes of moduleA, then we will get error.

```
C:\Users\Durga\Desktop>java --module-path out -m moduleB/pack2.Test
Error occurred during initialization of boot layer
java.lang.module.FindException: Module moduleA not found, required by moduleB
```

Use cases of Optional Dependencies:

Usage of optional dependencies is very common in Programming world.

Sometimes we can develop library with optional dependencies.

Eg 1: If apache http Client is available use it, otherwise use HttpURLConnection.
Eg 2: If oracle module is available use it, otherwise use mysql module.

Why we should do this? For various reasons –

1. When distributing a library and we may not want to force a big dependency to the client.
2. On the other hand, a more advanced library may have performance benefits, so whatever module client needs, he can use.
3. We may want to allow easily pluggable implementations of some functionality. We may provide implementations using all of these, and pick the one whose dependency is found.

Q. What is the difference between the following?

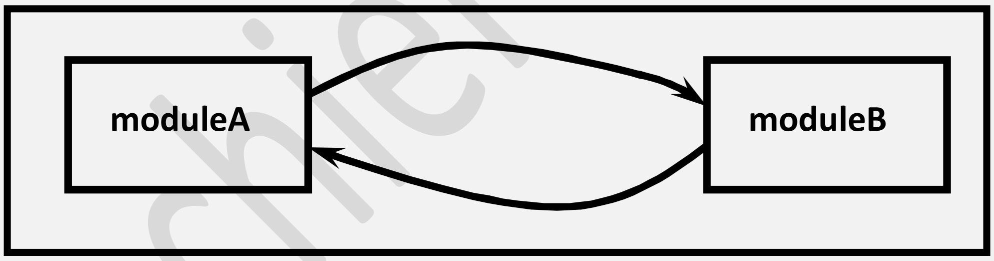
```
module  moduleB
{
    1. requires  moduleA;
    2. requires  transitive   moduleA
    3. requires  static   moduleA
 }
```

Cyclic Dependencies:

If moduleA depends on moduleB and moduleB depends on moduleA, such type of dependency is called cyclic dependency.

Cyclic Dependencies between the modules are not allowed in java 9.

Demo Program:



moduleA components:

module-info.java

```
module  moduleA
{
    requires  moduleB;
}
```

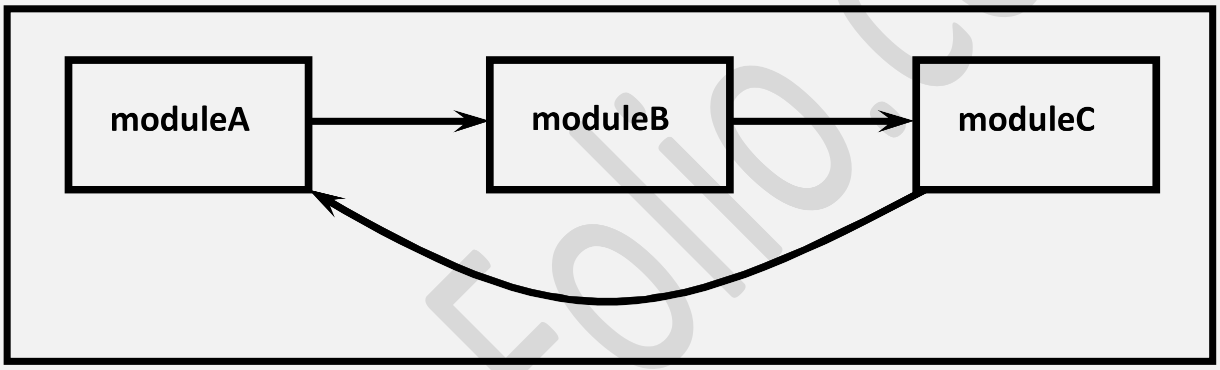
moduleB components:

module-info.java

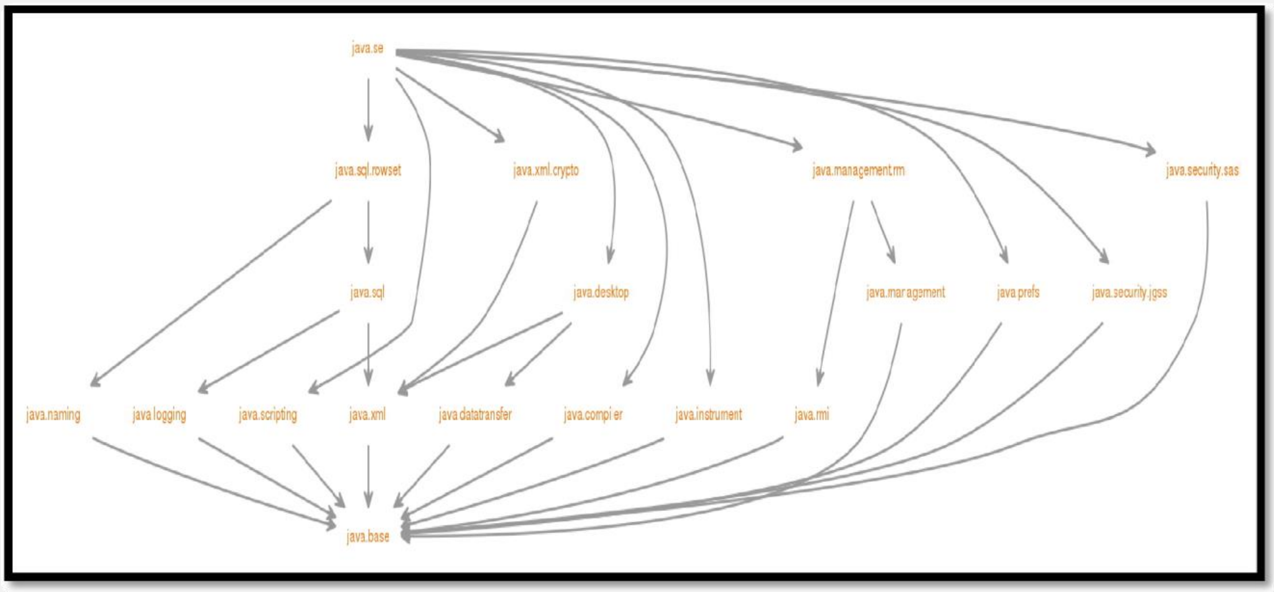
```
module  moduleB
{
    requires  moduleA;
}
```

```
C:\Users\Durga\Desktop>javac  --module-source-path src -d out -m moduleA, moduleB
src\moduleB\module-info.java:3: error: cyclic dependence involving moduleA
        requires moduleA;
```

There may be a chance of cyclic dependency between more than 2 modules also.
moduleA requires moduleB
moduleB requires moduleC
moduleC requires moduleA



Note: In all predefined modules also, there is no chance of cyclic dependency



Qualified Exports:

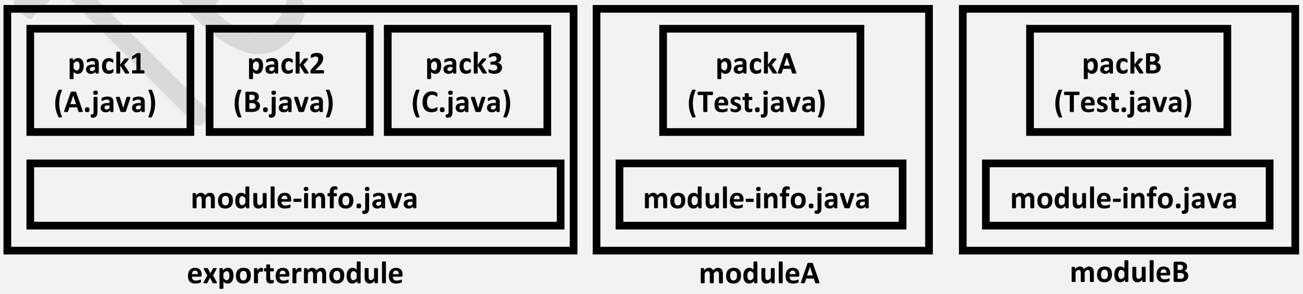
Sometimes a module can export its package to specific module instead of every module. Then the specified module only can access. Such type of exports are called Qualified Exports.

Syntax:
exports <pack1> to <module1>,<module2>,...

Eg:

```
module moduleA
{
    exports pack1; //to export pack1 to all modules
    exports pack1 to moduleA; // to export pack1 only for moduleA
    exports pack1 to moduleA,moduleB; // to export pack1 for both moduleA,moduleB
}
```

Demo Program for Qualified Exports:



Components of exportermodule:

A.java:

```java
package  pack1;
public  class  A
{
}
```

B.java:

```java
package  pack2;
public  class  B
{
}
```

C.java:

```java
package  pack3;
public  class  C
{
}
```

module-info.java:

```java
module   exportermodule
{
    exports  pack1;
    exports  pack2  to  moduleA;
    exports  pack3  to  moduleA,moduleB;
}
```

	moduleA	moduleB
pack1	√	√
pack2	√	□
pack3	√	√

Components of moduleA:

Test.java:

```java
package  packA;
import   pack1.A;
import   pack2.B;
import   pack3.C;
public  class  Test
{
    public  static  void  main(String[]  args)
    {
        System.out.println(  "Qualified  Exports  Demo");
    }
}
```

module-info.java:

```
module  moduleA
{
    requires   exportermodule;
}
```

Explanation:

For moduleA, all 3 packages are available. Hence we can compile and run moduleA successfully.

```
C:\Users\Durga\Desktop>javac --module-source-path src -d out -m exportermodule, moduleA

C:\Users\Durga\Desktop>java --module-path out -m moduleA/packA.Test
Qualified Exports Demo
```

Components of moduleB:

Test.java:

```
package  packB;
 import  pack1.A;
import pack2.B;
 import  pack3.C;
 public  class  Test
{
    public  static  void  main(String[]  args)
    {
        System.out.println(  "Qualified  Exports  Demo");
    }
}
```

module-info.java:

```
module  moduleB
{
    requires   exportermodule;
}
```

Explanation:

For moduleB, only pack1 and pack3 are available. pack2 is not available. But in moduleB we are trying to access pack2 and hence we will get compile time error.

```
C:\Users\Durga\Desktop>javac --module-source-path src -d out -m exportermodule, moduleB
src\moduleB\packB\Test.java:3: error: package pack2 is not visible
```

```
import pack2.B;
      ^
  (package pack2 is declared in module exportermodule, which does not export it to module moduleB)
1 error
```

Q. Which of the following directives are valid inside module-info.java:

1. requires moduleA;
2. requires moduleA,moduleB;
3. requires moduleA.pack1;
4. requires moduleA.pack1.A;
5. requires static moduleA;
6. requires transitive moduleA;
7. exports pack1;
8. exports pack1,pack2;
9. exports moduleA;
10. exports moduleA.pack1.A;
11. exports pack1 to moduleA;
12. exports pack1 to moduleA,moduleB;

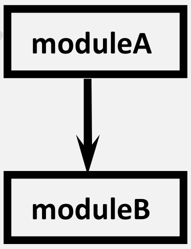
Answers: 1,5,6,7,11,12

Module Graph:

The dependencies between the modules can be represented by using a special graph, which is nothing but Module Graph.

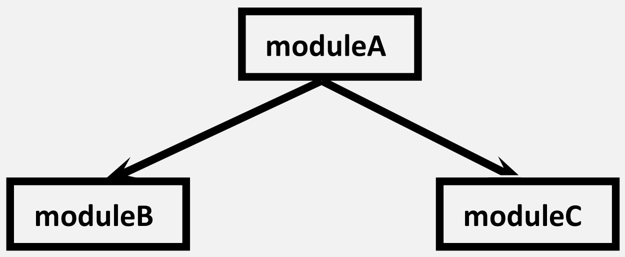
Eg1: If moduleA requires moduleB then the corresponding module graph is :

```
module  moduleA
{
     requires  moduleB;
}
```



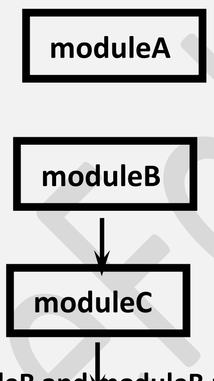
Eg 2: If moduleA requires moduleB and moduleC then the corresponding module graph is:

```
module  moduleA
{
     requires  moduleB;
     requires  moduleC;
}
```



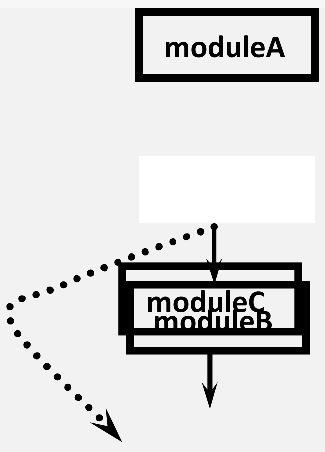
Eg 3: If moduleA requires moduleB and moduleB requires moduleC then the corresponding module graph is:

```
module  moduleA
{
    requires  moduleB;
}
module  moduleB
{
    requires  moduleC;
}
```



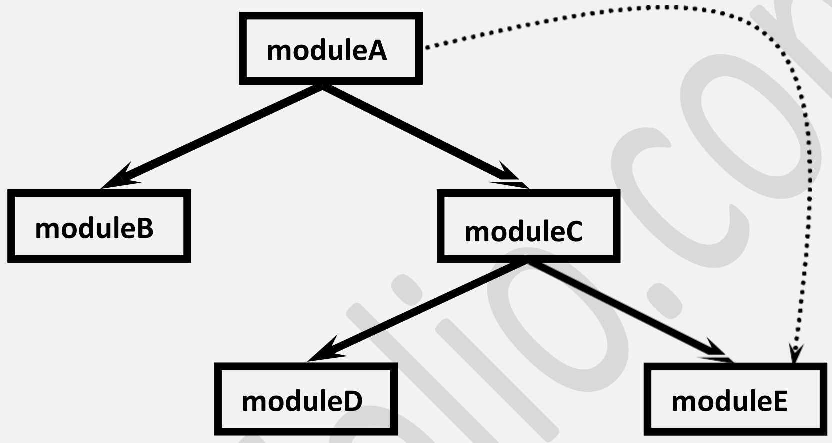
Eg 4: If moduleA requires moduleB and moduleB requires transitive moduleC then the corresponding module graph is:

```
module  moduleA
{
    requires  moduleB;
}
module  moduleB
{
    requires  transitive   moduleC;
}
```



<u>Eg 5:</u> If moduleA requires moduleB and moduleC, moduleC requires moduleD and transitive moduleE then the corresponding Modular Graph is:

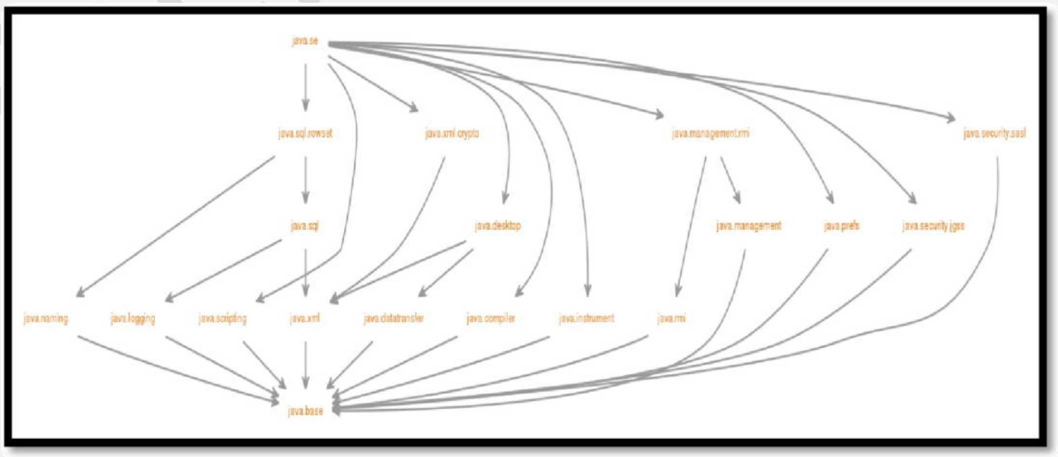
```
module  moduleA
{
     requires  moduleB;
     requires  moduleC;
}
module  moduleC
{
     requires  moduleD;
     requires  transitive   moduleE;
}
```



Java 9 JDK itself modularized. All classes of Java SE are divided into several modules.

<u>Eg:</u>
- **java.base**
- **java.sql**
- **java.xml**
- **java.rmi**
- **etc...**

The module graph of JDK is



In the above diagram all modules requires java.base module either directly or indirectly. Hence this module acts as BASE module for all java modules.
Observe modular graphs carefully:java.se and java.sql modules etc

Rules of Module Graph:

1. Two modules with the same name is not allowed.
2. Cyclic Dependency is not allowed between the modules and hence Module Graph should not contain cycles.

Observable Modules:

The modules which are observed by JVM at runtime are called Observable modules.

The modules we are specifying with --module-path option with java command are observed by JVM and hence these are observable modules.

```
java --module-path  out -m moduleA/pack1.Test
```

The modules present in module-path out are observable modules

JDK itself contains several modules (like java.base, java.sql, java.rmi etc). These modules can be observed automatically by JVM at runtime and we are not required to use --module-path. Hence these are observable modules.

Observable Modules = All Predefined JDK Modules + The modules specified with --module-path option

We can list out all Observable Modules by using --list-modules option with java command.

Eg1: **To print all readymade compiled modules (pre defined modules) present in Java 9**

```
C:\Users\Durga\Desktop>java --list-modules
java.activation@9
java.base@9
java.compiler@9
....
```

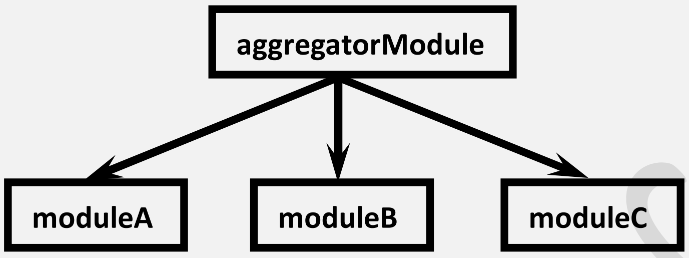
Eg2: **Assume our own created compiled modules are available in out folder. To list out these modules including readymade java modules**

```
C:\Users\Durga\Desktop>java --module-path out --list-modules
java.activation@9
java.base@9
...
exportermodule file:///C:/Users/Durga/Desktop/out/exportermodule/
moduleA file:///C:/Users/Durga/Desktop/out/moduleA/
```

Aggregator Module:

Sometimes a group of modules can be reused by multiple other modules. Then it is not recommended to read each module individually. We can group those common modules into a single module, and we can read that module directly. This module which aggregates functionality of several modules into a single module is called Aggregator module. If any module reads aggregator module then automatically all its modules are by default available to that module.

Aggregator module won't provide any functionality by its own, just it gathers and bundles together a bunch of other modules.



```
module  aggregatorModule
{
   requires  transitive   moduleA;
   requires  transitive   moduleB;
   requires  transitive   moduleC;
}
```

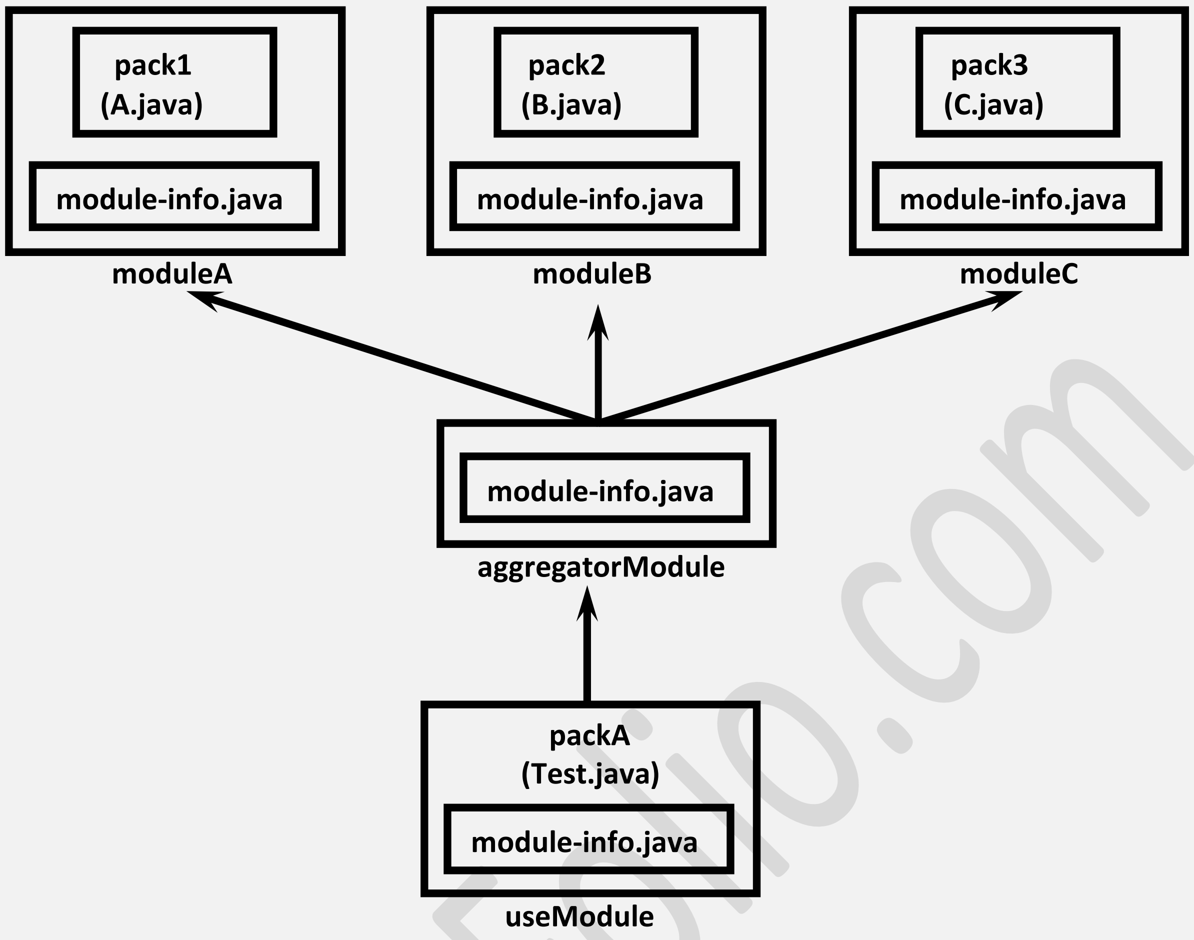
Aggregator Module not required to contain a single java class. Just it "requires transitive" of all common modules.

If any module reads aggregatorModule automatically all 3 modules are by default available to that module also.

```
module  useModule
{
   requires  aggregatorModule;
}
```

Now useModule can use functionality of all 3 modules moduleA, moduleB and moduleC.

Demo Program for Aggregator Module:



moduleA components:

A.java:

```java
package  pack1;
public  class  A
{
    public  void  m1()
    {
        System.out.println(  "moduleA  method" );
    }
}
```

module-info.java:

```java
module  moduleA
{
    exports  pack1;
}
```

moduleB components:

B.java:

```java
package  pack2;
public  class  B
{
    public  void  m1()
    {
        System.out.println(  "moduleB  method" );
    }
}
```

module-info.java:

```java
module  moduleB
{
    exports  pack2;
}
```

moduleC components:

C.java:

```java
package  pack3;
public  class  C
{
    public  void  m1()
    {
        System.out.println(  "moduleC  method" );
    }
}
```

module-info.java:

```java
module  moduleC
{
    exports  pack3;
}
```

aggregatorModule components:

module-info.java:

```java
module  aggregatorModule
{
    requires  transitive   moduleA;
    requires  transitive   moduleB;
    requires  transitive   moduleC;
}
```

useModule components:

Test.java:

```
package  packA;
 import  pack1.A;
import pack2.B;
 import  pack3.C;
 public  class  Test
{
    public  static  void  main(String[]  args)
    {
        System.out.println(  "Aggregator  Module  Demo");
        A a = new  A();
        a.m1();

        B b = new  B();
        b.m1();

        C c = new  C();
        c.m1();
    }
}
```

module-info.java:

Here we are not required to use requires directive for every module, just we have to use requires only for aggregatorModule.

```
 module  useModule
{
    //requires   moduleA;
    //requires   moduleB;
    //requires   moduleC;
    requires   aggregatorModule;
 }
```

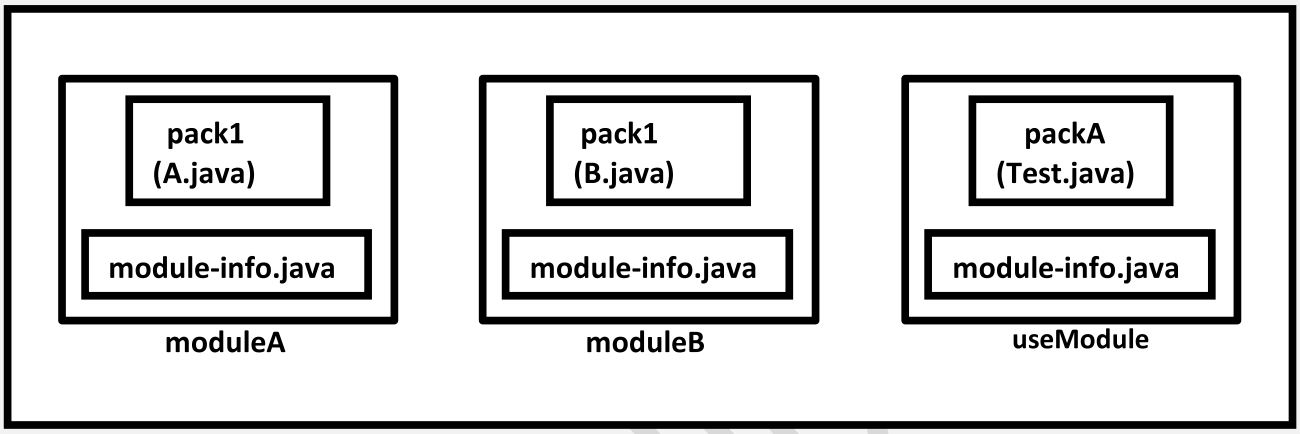
```
C:\Users\Durga\Desktop>javac --module-source-path src -d out -m
moduleA,moduleB,moduleC,aggregatorModule,useModule

C:\Users\Durga\Desktop>java --module-path out -m useModule/packA.Test
Aggregator Module Demo
moduleA method
moduleB method
moduleC method
```

Package Naming Conflicts:

Two jar files can contain a package with same name, which may creates version conflicts and abnormal behavior of the program at runtime.

But in Java 9 module System, two modules cannot contain a package with same name; otherwise we will get compile time error. Hence in module system, there is no chance of version conflicts and abnormal behavior of the program.

Demo Program:



A.java:

```
package  pack1;
public  class  A
{
    public  void  m1()
    {
        System.out.println(  "moduleA  method" );
    }
}
```

module-info.java:

```
module  moduleA
{
    exports  pack1;
}
```

moduleB components:

B.java:

```java
package pack1;
public class B
{
    public void m1()
    {
        System.out.println( "moduleB method" );
    }
}
```

module-info.java:

```java
module moduleB
{
    exports pack1;
}
```

useModule components:

Test.java:

```java
package packA;
public class Test
{
    public static void main(String[] args)
    {
        System.out.println( "Package Naming Conflicts" );
    }
}
```

module-info.java:

```java
module useModule
{
    requires moduleA;
    requires moduleB;

}
```

```
javac --module-source-path src -d out -m moduleA,moduleB,useModule
error: module useModule reads package pack1 from both moduleA and moduleB
```

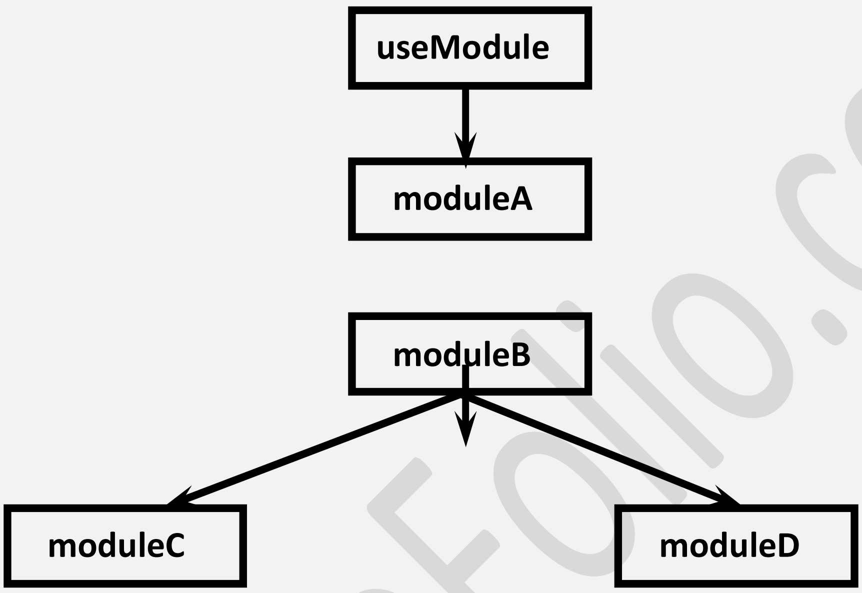
Two modules cannot contain a package with same name.

Module Resolution Process (MRP):

In the case of traditional classpath, JVM won't check the required .class files at the beginning. While executing program if JVM required any .class file, then only JVM will search in the classpath for the required .class file. If it is available then it will be loaded and used and if it is not available then at runtime we will get NoClassDefFoundError,which is not at all recommended.

But in module programming, JVM will search for the required modules in the module-path before it starts execution. If any module is missing at the beginning only JVM will identify and won't start its execution. Hence in modular programming, there is no chance of getting NoClassDefFoundError in the middle of program execution.

Demo Program:



Components of useModule:

Test.java:

```
package packA;
public class Test
{
    public static void main(String[]  args)
    {
        System.out.println(  "Module  Resolution  Process(MRP)  Demo");
    }
}
```

module-info.java:

```
module  useModule
```

```
{
    requires  moduleA;
}
```

Components of moduleA:

module-info.java:

```
module  moduleA
{
    requires  moduleB;
}
```

Components of moduleB:

module-info.java:

```
module  moduleB
{
    requires  moduleC;
    requires  moduleD;
}
```

Components of moduleC:

module-info.java:

```
module  moduleC
{

}
```

Components of moduleD:

module-info.java:

```
module  moduleD
{

}
```

```
javac --module-source-path src -d out -m moduleA,moduleB,moduleC,moduleD,useModule
```

```
java --module-path out --show-module-resolution -m useModule/packA.Test
```

The module what we are trying to execute will become root module.
Root module should contain the class with main method.

The main advantages of Module Resolution Process at beginning are:

1. We will get error if any dependent module is not available.
2. We will get error if multiple modules with the same name
3. We will get error if any cyclic dependency
4. We will get error if two modules contain packages with the same name.

Note:

The following are restricted keywords in java 9:
module,requires,transitive,exports
In normal java program no restrictions and we can use for identifier purpose also.